Midterm Exam Solution

```
    ____  ____  ___  ____    _______  __
   / ___/ ___/ ___/ ___/   /__  /__  / _ \
  / _/ / _/ / /   \__ \    /_ < / / / / /
 / /__/ /__/ /___ ___) /  ___) |/ / / /_/ /
/____/____/\____/____/  /____//_/  \___/
```

EECS 370 Spring 2023: Introduction to Computer Organization

You are to abide by the University of Michigan College of Engineering Honor Code. Please sign below to signify that you have kept the honor code pledge:

I have neither given nor received aid on this exam,
nor have I concealed any violations of the Honor Code.

Signature: ANSWER KEY

Name: ______________________

Uniqname: ______________________

Uniqname of person sitting to your ***Right***
(Write ⊥ if you are at the end of the row) ______________________

Uniqname of person sitting to your ***Left***
(Write ⊥ if you are at the end of the row) ______________________

Exam Directions:

- You have **120 minutes** to complete the exam. There are **7** questions in the exam on **15** pages (double-sided). **Please flip through your exam to ensure you have all pages.**
- You must show your work to be eligible for partial credit!
- Write legibly and dark enough for the scanners to read your answers.
- **Write your uniqname on the line provided at the top of each page.**

Exam Materials:

- You are allotted **one 8.5 x 11 double-sided** note sheet to bring into the exam room.
- You are allowed to use calculators that do not have an internet connection. All other electronic devices, such as cell phones or anything or calculators with an internet connection, are strictly forbidden.

Problem	1	2	3	4	5	6	7
Point Value	15	13	11	9	12	15	25

Problem 1: Multiple Choice 15 points

Completely shade in the boxes with the correct answers. Select only 1 answer, unless specified by "**(FILL IN ALL THAT APPLY).**" *[1.5 points each]*

1. Which of the following is usually true of RISC ISAs, in comparison to CISC ISAs? (**FILL IN ALL THAT APPLY)**
   - ☐ Produce smaller programs
   - [✓ ☐ Support fewer instructions]
   - ☐ Encode different instructions to different bit widths
   - [✓ ☐ Have simpler decoding logic]

2. The error "`program.o: unresolved external symbol 'int x'`" would be reported during which of the below processes?
   - ☐ Compilation
   - ☐ Assembly
   - [✓ ☐ Linking]
   - ☐ Loading
   - ☐ Execution

3. Endianness determines which of the below (assume a byte addressable system)? (**FILL IN ALL THAT APPLY)**
   - ☐ Which varieties of load/store instructions are supported in an ISA
   - ☐ Which bit read from a byte of memory is considered most significant
   - [✓ ☐ Which byte read from a word in memory is considered most significant]
   - ☐ Which bits of a register are used to hold a single byte loaded from memory

4. LC2K has what kind of addressability?
   - ☐ Byte
   - [✓ ☐ Word]
   - ☐ Big endian
   - ☐ Little endian

5. When assembling the following code LEGv8, what offset would the label `func` be replaced with?

```
        b.le    func
        add     X0,   X3,   X7
func    lsl     X2,   X4,   #3
```

   - ☐ 1
   - [✓ ☐ 2]
   - ☐ 4
   - ☐ 8
   - ☐ It depends on what line in the program the instructions are placed

6. Which LC2K instructions calculate a memory address by adding a base value to an offset? (**FILL IN ALL THAT APPLY)**

   - ☐ `add`
   - [✓ ☐ `lw`]
   - [✓ ☐ `sw`]
   - ☐ `jalr`
   - [✓ ☐ `beq`]

7. Increasing the number of registers in an ISA, while keeping the instruction size the same, generally: (**FILL IN ALL THAT APPLY)**

   - [✓ ☐ Decreases the number of loads and stores needed to execute a given program]
   - ☐ Increases the number of opcodes that can be supported
   - [✓ ☐ Requires more bits be used to indicate each register]
   - ☐ Results in lower ALU latency

8. What is the effect of running the following 2-line LC2K program?

```
        .fill    3
        halt
```

   - ☐ Nothing happens
   - ☐ A value is written into memory
   - [✓ ☐ A value is written into a register]
   - ☐ The program infinite loops
   - ☐ The program has undefined behavior

9. Moving from a single-cycle processor to a multi-cycle processor is expected to reduce which of the following when running a given program? (**FILL IN ALL THAT APPLY)**

   - [✓ ☐ Cycle time]
   - ☐ Number of instructions executed
   - ☐ Average cycles to execute each instruction
   - [✓ ☐ Average time to execute some instructions]

10. Which of the following can be implemented without sequential logic? (**FILL IN ALL THAT APPLY)**

   - [✓ ☐ Mux]
   - ☐ PC
   - [✓ ☐ ALU]
   - [✓ ☐ Decoder]
   - ☐ Register

Problem 2: Mixed Signals 13 points

1. Convert each of the 8-bit hexadecimal numbers into its binary form and decimal form (both for treating the original number as a signed (two's-complement) and an unsigned value). *[2.5 points]*

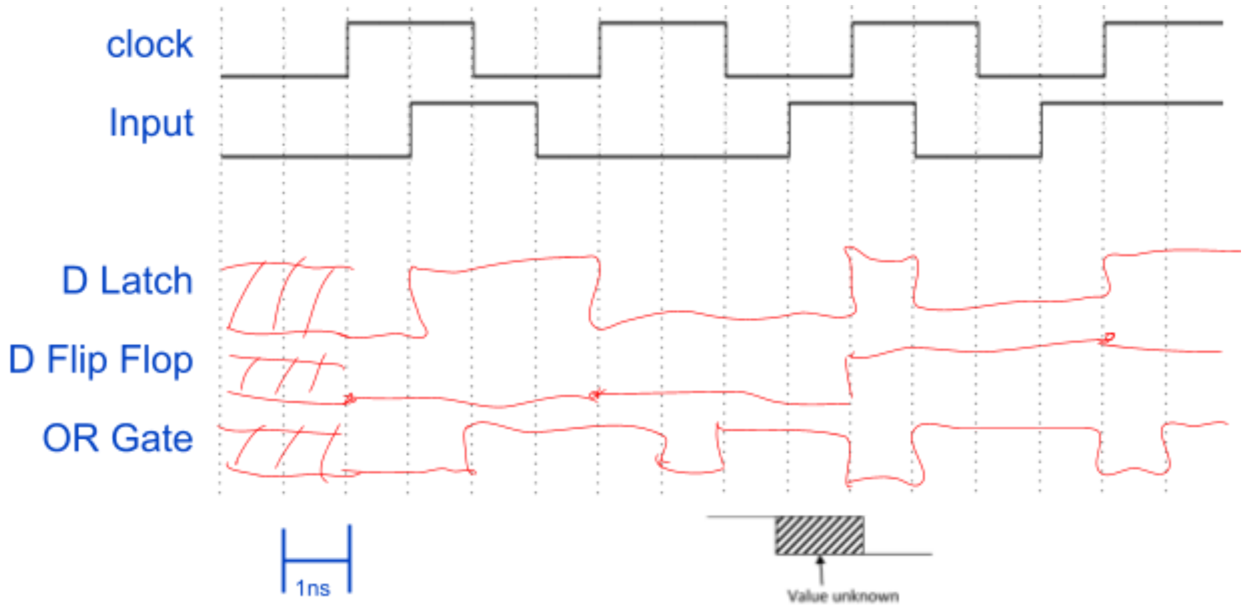
Hexadecimal	Binary	Decimal (signed)	Decimal (unsigned)
0x2C	0b0010 1100	44	44
0x98	0b1001 1000	-104	152

2 * 16 + 12 = 44

-128 + 16 + 8 = -152

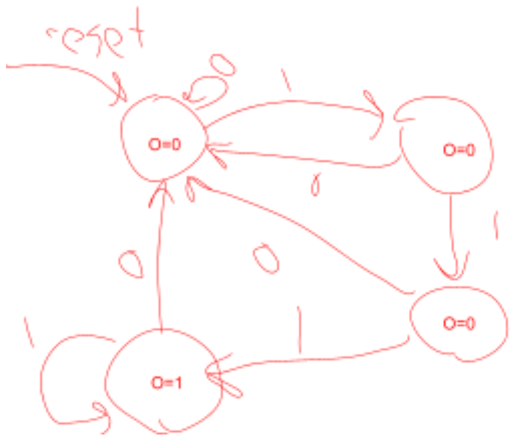
128 + 16 + 8 = 104

2. Complete the timing diagram below for a D latch, a rising-edge triggered D flip-flop, and a delayed OR gate.
   - In the case of the latch, "clock" is the Gate signal.
   - Assume there is no meaningful delay for the latch and flip-flop.
   - In the case of the OR gate, "clock" is the other input and the gate has a delay of **2ns**.
   - If a value is unknown, indicate that clearly using the notation shown. *[3.5 points]*

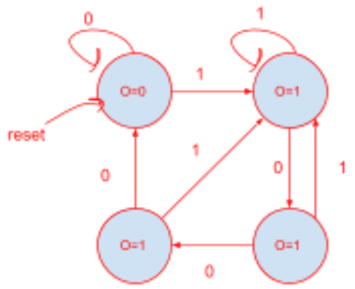


3. Draw a Moore-style finite state machine with one input (I) and one output (O). O should be set to 1 if and only if I had been set to 1 the previous 3 cycles. Write the value of "I" that triggers each transition next to the corresponding arrow, and the output of "O" where appropriate. Your state machine must have the minimum number of states needed to receive full credit. Don't forget to indicate a reset state. *[3.5 points]*

Intended interpretation (if input was high for ALL of the past 3 cycles, output 1)



Alternative interpretation (if input was high for ANY of the past 3 cycles, output 1)



4. Consider an 8-bit floating point format based on the IEEE standard where the most significant is used for the sign, the next 3 bits are used for the exponent and the last 4 bits are used for the mantissa. The scheme uses "biased 3" to represent the exponent (rather than biased 127 used for a 32-bit IEEE floating point number) and has an implicit one just like the IEEE format. This scheme is called "VSF" (very short float).

Sign	Exponent	Mantissa
7	6 5 4	3 2 1 0

Write the *binary* encoding of -0.75 as a VSF number. *[3.5 points]*

0b___10101000___________

0.75 = ½ + ¼.
½ will be the implicit 1. So mantissa is 0b1000
0.75 = 1.5 * 2^{-1}, so encoded exponent is -1 + 3 = 0b010

Problem 3: Rough Root 11 points

Consider the following C function (on the left) that takes as input a non-negative integer n and returns the floor of its square root. A programmer translated it into ARM by hand, but unfortunately made some mistakes. Answer the questions on the next page. **Assume the mul instruction is supported and is used correctly on line 13.**

	C (correct, tested)		LEGv8 (buggy, hand-written)	
1	`unsigned flrsqrt(unsigned n) {`	1	`flrsqrt:`	1
2	`   unsigned r = n;`	2	`        mov     X10, X0`	2
3	`   unsigned l = 0;`	3	`        mov     X5,  X10`	3
4	`   unsigned ans = 0;`	4	`        mov     X4,  #0`	4
5	`   while ( l <= r ) {`	5	`        mov     X6,  #0`	5
6		6	`        b       comp`	6
7	`       unsigned m = (l + r) / 2;`	7	`loop:`	7
8	`       unsigned m2 = m * m;`	8	`        cmp     X4, X5`	8
9		9	`        b.gt    end`	9
10	`       if (m2 == n) {`	10	`body:`	10
11	`           return m;`	11	`        add     X3, X4, X5`	11
12		12	`        lsr     X7, X3, #1`	12
13	`       } else if (m2 < n) {`	13	`        mul     X8, X7, X7`	13
14		14	`        cmp     X8, X10`	14
15	`           l = m + 1;`	15	`        b.ne    elif`	15
16	`           ans = m;`	16	`        mov     X0, X7`	16
17		17	`        b       ret`	17
18	`       } else {`	18	`elif:`	18
19	`           r = m - 1;`	19	`        cmp     X8, X10`	19
20		20	`        b.le    else`	20
21	`       }`	21	`        add     X4, X7, #1`	21
22	`   }`	22	`        mov     X6, X7`	22
23	`   return ans;`	23	`        b       loop`	23
24	`}`	24	`else:`	24
25		25	`        sub     X5, X7, #1`	25
26		26	`        b       loop`	26
27		27	`end:`	27
28		28	`        mov     X0, X7`	28
29		29	`ret:`	29
30		30	`        br      lr`	30

a) Below is a list of the local variables declared in the C function. What registers did the programmer choose to store these values? Which is used as a scratch register? *[7 points]*

Variable/Scratch	Register
`n`	X10 (½ credit for X0)
`r`	X5
`l`	X4
`ans`	X6
`m`	X7
`m2`	X8
`<scratch>`	X3

b) There are two bugs after line 18 - one is an incorrect opcode, the other is an incorrect source register. Identify the two bugs and suggest a one line fix for each. *[4 points]*

Bug 1
Line #: 20
Replacement instruction: `b.ge else`

Bug 2
Line #: 28
Replacement instruction: `mov X0, X6`

Problem 4: Some Assembly Required 9 points

Fill in the blanks of the object file. Recall that the 0x notation indicates that the answer is expected in *hexadecimal.* (That only applies to the blanks that start with 0x, all other answers should be in the format expected in project 2a.)

PC	main.as	PC	main.obj
			21 2 7 12
0	Main lw 0 1 Count	0	8454144
1	loop lw 0 6 One	1	8781845
2	sw 5 1 Stack	2	15269888
3	add 5 6 5	3	3014661
4	lw 0 6 Func1	4	0x860000
5	jalr 6 7	5	24576000
6	lw 0 6 Neg1	6	8781824
7	add 5 6 5	7	3014661
8	lw 5 1 Stack	8	11075584
9	lw 0 6 One	9	8781845
10	sw 5 1 Stack	10	15269888
11	add 5 6 5	11	3014661
12	lw 0 6 func2	12	8781846
13	jalr 6 7	13	24576000
14	lw 0 6 Neg1	14	8781824
15	add 5 6 5	15	3014661
16	lw 5 1 Stack	16	11075584
17	add 1 6 1	17	917505
18	beq 1 0 end	18	0x1080001
19	beq 0 0 loop	19	16842733
20	end halt	20	25165824
21	One .fill 1	21	1
22	func2 .fill Delta	22	0
			Main T 0
			One D 0
			Stack U 0
			Count U 0
			Func1 U 0
			Neg1 U 0
			Delta U 0
			0 lw Count
			1 lw One
			1 .fill Delta
			2 sw Stack
			4 lw Func1
			6 lw Neg1
			8 lw Stack
			9 lw One
			10 sw Stack
			12 lw func2
			14 lw Neg1
			16 lw Stack

Problem 5: Call-ee-2-Save 12 points

Consider the following two LC2K files and answer the questions below.

main.as (repeated from last problem)

```
Main    lw      0     1     Count
loop    lw      0     6     One
        sw      5     1     Stack
        add     5     6     5
        lw      0     6     Func1
        jalr    6     7
        lw      0     6     Neg1
        add     5     6     5
        lw      5     1     Stack
        lw      0     6     One
        sw      5     1     Stack
        add     5     6     5
        lw      0     6     func2
        jalr    6     7
        lw      0     6     Neg1
        add     5     6     5
        lw      5     1     Stack
        add     1     6     1
        beq     1     0     end
        beq     0     0     loop
end     halt
One     .fill   1
func2   .fill   Delta
```

funcs.as

```
alpha   lw      0     6     One
        sw      5     2     Stack
        add     5     6     5
        sw      5     4     Stack
        add     5     6     5
        nor     1     1     2
        add     2     1     4
        nor     2     4     3
        lw      0     6     Neg1
        add     5     6     5
        lw      5     4     Stack
        add     5     6     5
        lw      5     2     Stack
        jalr    7     6
Delta   lw      0     6     One
        sw      5     2     Stack
        add     5     6     5
        lw      0     3     Neg1
        add     5     3     5
        lw      5     2     Stack
        jalr    7     6
Count   .fill   3
Func1   .fill   alpha
Neg1    .fill   -1
```

Determine whether each of the following registers is caller / callee save ~~and the number of load/store pairs that occur when the program is executed~~.

Register	**Caller or Callee** *[3 pts each]*
1	Caller (2x/iter.)
2	Callee (2x/iter. in alpha and Delta)
4	Callee (1x/iter. in alpha)
6	Caller (Not saved since not live)

Problem 6: Addi it up 15 points

In this problem, you will implement some C code using a new ISA. The specifications for the ISA are below.

- Like LC2K, there are 8 registers, in the range [0-7]
- Registers and Instructions are 16 bits
- Immediates are 6-bit 2's complement numbers
- **Memory is byte addressable**
- Memory accesses must be aligned

Instruction Type	Parameters	Example
R (Register)	regA regB destReg	`nand 1 2 3`
I (Immediate)	regA regB immediate	`lw 0 1 label`
O	<none>	`halt`

Here is a subset of instructions relevant for the following problem:

Name	type	description
`nand`	R	destReg = ~(regA & regB)
`lsl`	R	destReg = regA << regB
`addi`	I	regB = regA + immediate
`lh`	I	regB = mem[regA + immediate] (2 bytes transferred)
`beq`	I	Branch to PC + 2 + immediate if regA == regB
`ret`	O	Returns to calling function - return address read from special purpose register

The following code snippet includes a function `score_attendance`, which is used to determine the number of labs attended by a student. There are 16 labs in the term, with each lab attendance grade corresponding to a bit in `attendance`.

1. Fill in the blanks to complete the implementation for score_attendance as a standalone function that stores the return value `sum` in register 7. At the beginning of the function execution, you may assume that register 0 contains 0, and register 1 contains `id`. All registers are caller-saved. Only use instructions from the above subset. *[12 points]*

```
#include <stdint.h>
#define CLASS_SIZE 64

struct student {
    uint8_t credits; // unused
    uint16_t attendance;
};

struct student class[CLASS_SIZE];

/* returns sum of attendances for
one student across all 16 labs */

uint16_t score_attendance (
    uint16_t id) {

    uint16_t sum = 0;

    uint8_t labNum = 16;
    while(labNum != 0) {
        labNum--;
        if (class[id].attendance
            & (1 << labNum)) {
            sum++;
        }
    }

    return sum;
}
```

```
            # init sum and labNum
            addi  0   7   #0
            addi  0   2   #16
loop        beq   0   2   end
            addi  2   2   #-1
            # load attendance
            addi  0   3   #2
            lsl   1   3   3
            addi  3   4   #2
            lh    4   5   class
            # apply mask
            addi  0   6   #1
            lsl   6   2   6
            nand  6   5   6
            nand  6   6   6
            beq   6   0   endif /
                      2 / loop / -22
            addi  7   7   #1
            # next iteration
endif       beq   0   0   loop
end         ret
class       .fill ...
```

2. When assembling the following code using this new ISA, what numeric immediate should the assembler replace the label "target" with? *[2 points]*

```
a          beq   0  0  target  0
a+1        add   0  0  0
a+2 target ret
```

immediate =

2

Problem 7: Just a Bit Clearer 25 points

Many versions of the ARM ISA provide a BIC, or Bit Clear, instruction that operates similar to the “reset” signal in an SR latch, clearing selected bits to 0 and leaving others unmodified. After seeing its great utility, we want to implement this instruction in LC2K.

In LC2K, `bic` will be an I-type instruction. For each bit that is a 1 in the contents of register A or in the memory value pointed to by offset, the corresponding bit in the contents of register B will be reset to 0.

We can represent this operation with the following C code:

```
regB = regB & ~(regA | Mem[offset]);
```

By DeMorgan’s laws, this can also be represented as:

```
regB = regB & ~regA & ~Mem[offset];
```

Or:

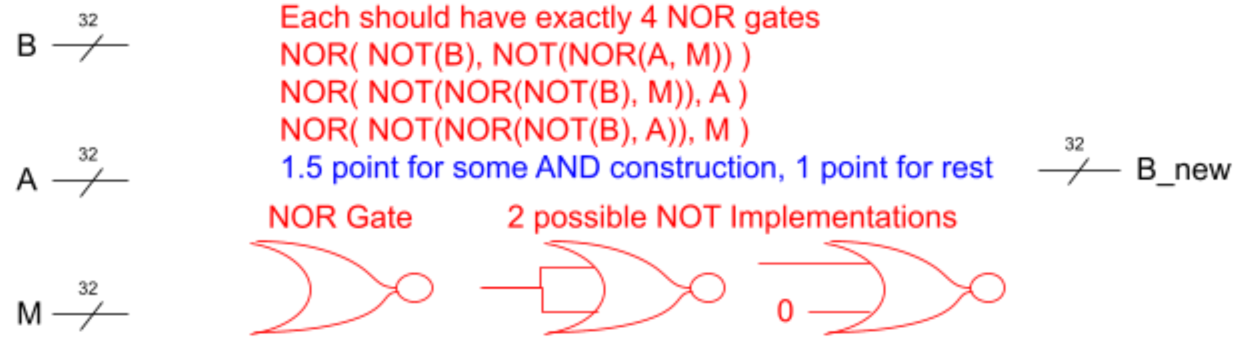
```
regB = ~(~regB | regA | Mem[offset]);
```

For example, the following `bic` with the corresponding `.fill` would clear bits 7-4 of register 1:

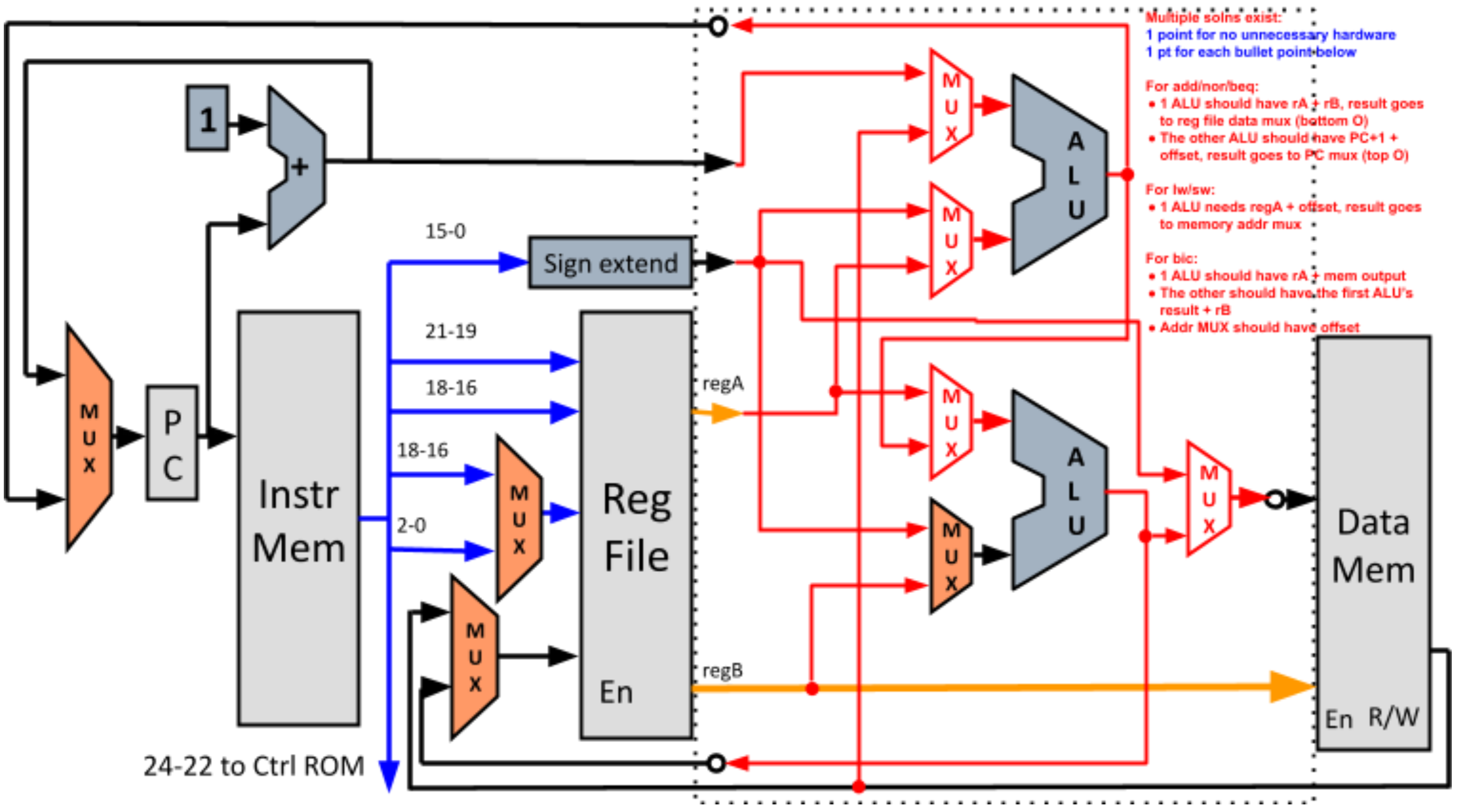
```
           bic       0    1    bmask
           …
bmask      .fill     240  //0xF0
```

a) To help you understand the data operations in this instruction, express `bic` with *only* 2-input bitwise NOR gates. Assume A is the contents of register A, B is the contents of register B, and M is the contents in memory pointed to by offset. *[2.5 points]*

Multiple Solutions Exist:
Each should have exactly 4 NOR gates
NOR(NOT(B), NOT(NOR(A, M)))
NOR(NOT(NOR(NOT(B), M)), A)
NOR(NOT(NOR(NOT(B), A)), M)
1.5 point for some AND construction, 1 point for rest



b) To perform part of the `bic` operation, the ALU has been extended to support bitwise **AND**, as well as the original addition and bitwise NOR. Modify the single-cycle datapath to support `bic` and **ALL** original LC2K instructions except `jalr and halt` with minimal hardware. You may add MUXes and connections inside the dotted box. You do not need to show any control signals. *[7 points]*



Multiple solns exist:
1 point for no unnecessary hardware
1 pt for each bullet point below

For add/nor/beq:
- 1 ALU should have rA + rB, result goes to reg file data mux (bottom O)
- The other ALU should have PC+1 + offset, result goes to PC mux (top O)

For lw/sw:
- 1 ALU needs regA + offset, result goes to memory addr mux

For bic:
- 1 ALU should have rA + mem output
- The other should have the first ALU's result + rB
- Addr MUX should have offset

c) After analyzing the original single-cycle datapath and your modified datapath, we have found that the clock period for the original design was 80ns and the clock period for the modified design is 144ns. We have a large program where X% of the instructions are `bic` instructions. To compare the designs, we have another version of the program that replaces all `bic` instructions with **5** other instructions, which we run on the original design.

1 point for comparing 2 equations, with 80ns on 1 side and 144 on the other side.

1 point for correct equation: 144 = 80(1 + 4X)

1 point for finding X = 0.2 = 20%. 1 point for circling "More"

Circle one & fill in the blank: [✓ More] / less than 20% of the instructions executed in the program must be `bic` instructions for the program to run faster on the modified design. Show your work above. *[4 points]*

d) It turns out that very few programs satisfy the constraint you found in part (c). To get better performance, a multi-cycle design might be better. The diagram (**provided as a separate reference sheet**) shows the multi-cycle datapath, modified with new connections to implement the `bic` instruction. As in part (b), the ALU supports bitwise **AND** with control signals 0b10.

Provide a sequence of operations that implement the `bic` instruction in 6 cycles. You might not use all blanks. Then fill out the empty boxes in the control ROM for the `bic` instruction. *[7.5 points]*

[0.7 points per cycle description, 0.4 pts per non-X box in the control ROM, 0.2 points for not writing to mem]

Cycle 1 (Instruction Fetch):	Instr Reg = Mem[PC],	ALU_result = PC + 1
Cycle 2 (Instruction Decode):	PC = ALU_result,	Read Register Values
Cycle 3:	DATA_reg = Mem[offset]	
Cycle 4:	ALU_result = NOR(Register A, DATA_reg)	
Cycle 5:	ALU_result = AND(ALU_result, Register B)	
Cycle 6 (Writeback):	Register B = ALU_result	

`bic` Cycle	PC en	MUX addr	MEM en	MEM r/w	IR en	MUX dest	MUX data	RFile wr en	MUX alu 1	MUX alu 2	ALU op
1 (IF)	0	00	1	0	1	X	X	0	00	001	00
2 (ID)	1	XX	0	X	0	X	X	0	XX	XXX	XX
3	0	10	1	0	0	X	X	0	XX	XXX	XX
4	0	XX	0	X	0	X	X	0	01	100	01
5	0	XX	0	X	0	X	X	0	10	000	10
6 (WB)	0	XX	0	X	0	0	1	1	XX	XXX	XX

e) Assume that our large program has the following distribution of instructions executed:

- 30% `add` instructions
- 5% `nor` instructions
- 10% `lw` instructions
- 25% `sw` instructions
- 15% `beq` instructions
- 15% **`bic`** instructions
- The program is large enough so that the `halt` instruction is negligible.

Find the CPI of this program on the new multi-cycle design, assuming that for the original LC2K instructions, the new design behaves exactly like the one described in class. Write down an exact decimal number for your answer. Show your work. *[4 points]*

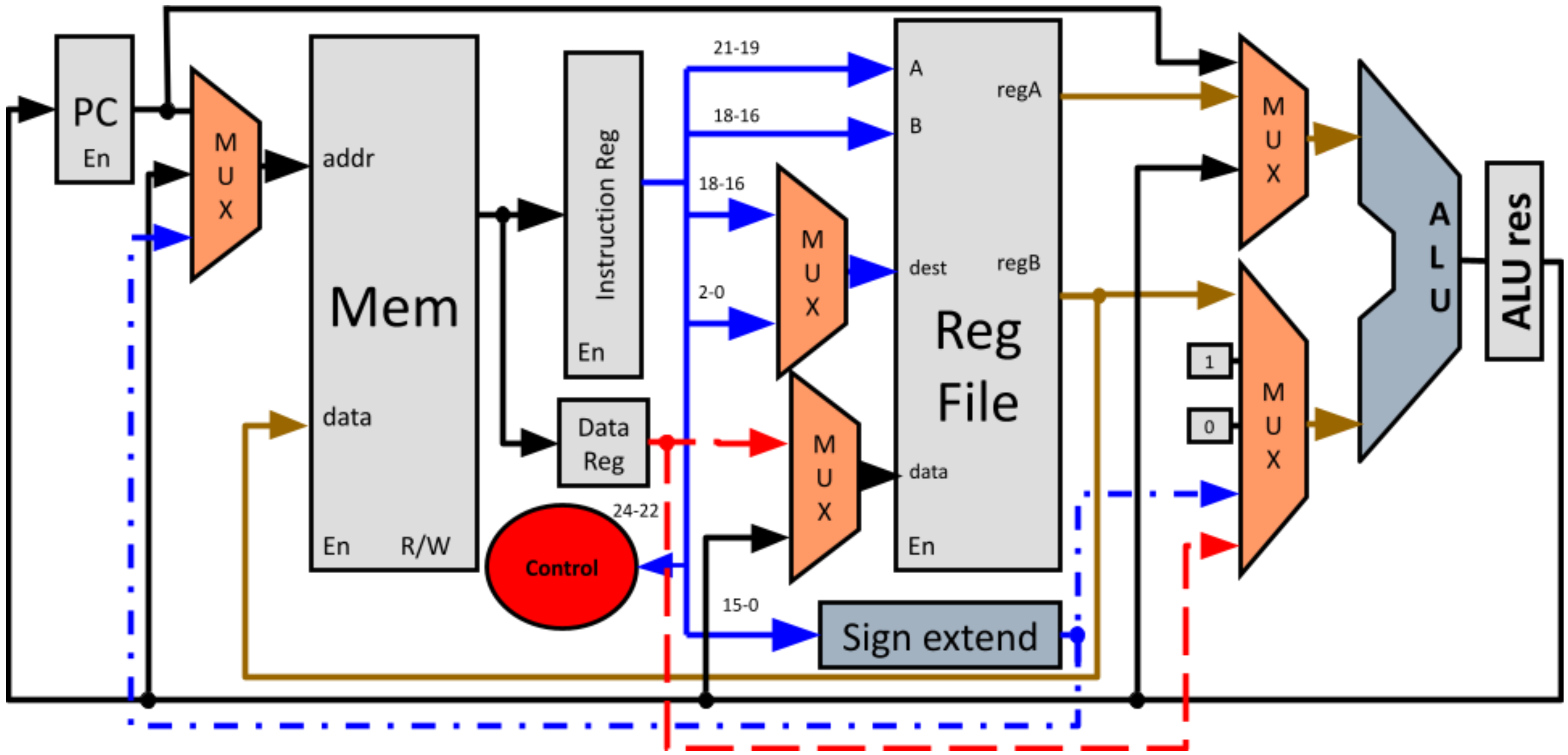
4 cycle instructions: 4 * (0.3 (add) + 0.05 (nor) + 0.25 (sw) + 0.15 (beq)) = 4 * 0.75 = 3

5 cycle instructions: 5 * 0.1 (lw) = 0.5

6 cycle instructions: 6 * 0.15 (bic) = 0.9

1 point for each of the above, 1 point for the final answer

CPI = 4.4


PC
En
M U X
addr
Mem
data
En
R/W
Instruction Reg
En
Data Reg
Control
24-22
21-19
18-16
18-16
2-0
15-0
M U X
M U X
A
B
dest
data
regA
regB
Reg File
En
Sign extend
M U X
M U X
1
0
ALU
ALU res